EXPLODE: a Lightweight, General System for Finding Serious Storage System Errors

Junfeng Yang, Can Sar, and Dawson Engler
Computer Systems Laboratory
Stanford University

Abstract

Storage systems such as file systems, databases, and RAID systems have a simple, basic contract: you give them data, they do not lose or corrupt it. Often they store the only copy, making its irrevocable loss almost arbitrarily bad. Unfortunately, their code is exceptionally hard to get right, since it must correctly recover from any crash at any program point, no matter how their state was smeared across volatile and persistent memory.

This paper describes EXPLODE*, a system that makes it easy to systematically check real storage systems for errors. It takes user-written, potentially system-specific checkers and uses them to drive a storage system into tricky corner cases, including crash recovery errors.* EXPLODE *uses a novel adaptation of ideas from model checking, a comprehensive, heavyweight formal verification technique, that makes its checking more systematic (and hopefully more effective) than a pure testing approach while being just as lightweight.*

EXPLODE *is effective. It found serious bugs in a broad range of real storage systems (without requiring source code): three version control systems, Berkeley DB, an NFS implementation, ten file systems, a RAID system, and the popular VMware GSX virtual machine. We found bugs in every system we checked, 36 bugs in total, typically with little effort.*

1 Introduction

Storage system errors are some of the most destructive errors possible. They can destroy persistent data, with almost arbitrarily bad consequences if the system had the only copy. Unfortunately, storage code is simultaneously both difficult to reason about and difficult to test. It must always correctly recover to a valid state if the system crashes at *any* program point, no matter what data is being mutated, flushed (or not flushed) to disk, and what invariants have been violated. Further, despite the severity of storage system bugs, deployed testing methods remain primitive, typically a combination of manual inspection (with the usual downsides), fixes in reaction to bug reports (from angry users) and, at advanced sites, the alleged use of manual extraction of power cords from sockets (a harsh test indeed, but not comprehensive).

This paper presents EXPLODE, a system that makes it easy to thoroughly check real systems for such crash recovery bugs. It gives clients a clean framework to build and plug together powerful, potentially system-specific dynamic storage checkers. EXPLODE makes it easy for checkers to find bugs in crash recovery code: as they run on a live system they tell EXPLODE when to generate the disk images that could occur if the system crashed at the current execution point, which they then check for errors.

We explicitly designed EXPLODE so that clients can check complex storage stacks built from many different subsystems. For example, Figure 1 shows a version control system on top of NFS on top of the JFS file system on top of RAID. EXPLODE makes it quick to assemble checkers for such deep stacks by providing interfaces that let users write small checker components and then plug them together to build many different checkers.

Checking entire storage stacks has several benefits. First, clients can often quickly check a new layer (sometimes in minutes) by reusing consistency checks for one layer to check all the layers below it. For example, given an existing file system checker, if we can slip a RAID layer below the file system we can immediately use the file system checker to detect if the RAID causes errors. (Section 9 uses this approach to check NFS, RAID, and a virtual machine.) Second, it enables strong end-to-end checks, impossible if we could only check isolated subsystems: correctness in isolation cannot guarantee correctness in composition [22]. Finally, users can localize errors by cross-checking different implementations of a layer. If NFS works incorrectly on seven out of eight file systems, it probably has a bug, but if it only breaks on one, that single file system probably does (§9.2).

We believe EXPLODE as described so far is a worthwhile engineering contribution. A second conceptual contribution is its adaptation of ideas from model checking [6, 15, 17], a typically heavyweight formal verification technique, to make its checking more systematic (and thus hopefully more effective) than a pure testing approach while remaining as lightweight as testing.

Traditional model checking takes a specification of a system (a "model") which it checks by starting from an initial state and repeatedly performing all possible actions to this state and its successors. A variety of techniques exist to make this exponential search less inefficient. Model checking has shown promise in finding

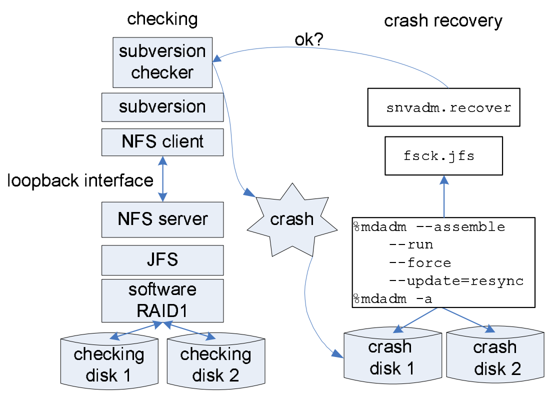


Figure 1: A snapshot of EXPLODE with a stack of storage systems being checked on the left and the recovery tools being run on the right after EXPLODE "crashes" the system to generate possible crash disks. This example checks Subversion running on top of NFS exporting a JFS file system running on RAID.

corner-case errors. However, requiring implementors to rewrite their system in an artificial modeling language makes it extremely expensive for typical storage systems (read: almost always impractical).

Recent work on *implementation-level model checking* [3, 13, 18] eliminates the need to write a model by using code itself as its own (high-fidelity) model. We used this approach in prior work to find serious errors in Linux file systems [30]. However, while more practical than a traditional approach, it required running the checked Linux system inside the model checker itself as a user-space process, which demanded enormously invasive modifications. The nature of the changes made it hard to check anything besides file systems and, even in the best case, checking a new file system took a week's work. Porting to a new Linux kernel, much less a different operating system, could take months.

This paper shows how to get essentially all the model checking benefits of our prior work with little effort by turning the checking process inside out. Instead of shoehorning the checked system inside the model checker (or worse, cutting parts of the checked system out, or worse still, creating models of the checked code) it interlaces the control needed for systematic state exploration *in situ*, throughout the checked system, reducing the modifications needed down to a single device driver, which can run inside of a lightly-instrumented, stock kernel running on real hardware. As a result, EXPLODE can thoroughly check large amounts of storage system code with little effort.

Running checks on a live, rather than emulated, system has several nice fallouts. Because storage systems already provide many management and configuration utilities, EXPLODE checkers can simply use this pre-built machinery rather than re-implementing or emulating it. It also becomes trivial to check new storage systems: just mount and run them. Finally, any check that can be run on the base system can also be run with EXPLODE.

The final contribution of the paper is an experimental evaluation of EXPLODE that shows the following:

1. EXPLODE checkers are effective (§7—§9). We found bugs in every system we checked, 36 bugs in total, typically with little effort, and often without source code (§8.1, §9.3). Checking without source code is valuable, since many robust systems rely on third-party software that must be vetted in the context of the integrated system.
2. EXPLODE checkers have enough power to do thorough checks, demonstrated by using it to comprehensively check ten Linux file systems (§7).
3. Even simple checkers find bugs (§8). Tiny checkers found bugs in three version control systems (§8.1) and a widely-used database (§8.2).
4. EXPLODE makes it easy to check subsystems designed to transparently slip into storage stacks (§9). We reused file system checkers to quickly find errors in RAID (§9.1), NFS (§9.2), and VMware (§9.3), which should not (but do) break the behavior of storage systems layered above or below them.

The paper is organized as follows. We first state our principles (§2) and then show how to use EXPLODE to check an example storage system stack (§3). We then give an overview of EXPLODE (§4) and focus on how it: (1) explores alternative actions in checked code (§5) and (2) checks crashes (§6). After the experimental evaluation (§7—§9), we discuss our experiences porting EXPLODE to FreeBSD (§ 10), contrast with related work (§11), and then conclude (§12).

2 Principles

In a sense, this entire paper boils down to the repeated application of a single principle:

Explore all choices: When a program point can legally do one of `N` different actions, fork execution `N` times and do each. For example, the kernel memory allocator can return `NULL`, but rarely does so in practice. For each call to this allocator we want to fork and do both actions. The next principle feeds off of this one:

Exhaust states: Do every possible action to a state before exploring another state. In our context, a state is defined as a snapshot of the system we check.

We distilled these two principles after several years of using model checking to find bugs. Model checking has a variety of tricks, some exceptionally complex. In retrospect, these capture the one feature of a model checking approach that we would take over all others: systematically do every legal action to a state, missing nothing, then pick another state, and repeat. This approach reliably finds interesting errors, even in well-tested code. We are surprised when it does not work. The key feature of this principle over traditional testing is that it makes low-probability events (such as crashes) as probable as high-probability events, thereby quickly driving the checked system into tricky corner-cases. The final two principles come in reaction to much of the pain we had with naive application of model checking to large, real systems.

Touch nothing. Almost invariably, changing the behavior of a large checked system has been a direct path to experiences that we never want to repeat. The internal interfaces of such systems are often poorly defined. Attempting to emulate or modify them produces corner-case mistakes that model checking is highly optimized to detect. Instead we try to do everything possible to run the checked system as-is and parasitically gather the information we need for checking as it runs.

Report only true errors, deterministically. The errors our system flags should be real errors, reduced to deterministic, replayable traces. All checking systems share this motherhood proclamation, but, in our context it has more teeth than usual: diagnosing even deterministic, replayable storage errors can take us over a day. The cost of a false one is enormous, as is the time needed to fight with any non-determinism.

3 How to Check a Storage System

This section shows how clients use EXPLODE interfaces to check a storage system, using a running example of a simple file system checker. Clients use EXPLODE to do two main things to a storage system. First, systematically exhaust all possibilities when the checked system can do one of several actions. Second, check that it correctly recovers from a crash. Clients can also check non-crash properties by simply inserting code to do so in either their checker or checked code itself without requiring EXPLODE support (for an example see §7.2).

Below, we explain how clients expose decision points in the checked code (§ 3.1). We then explain the three system-specific components that clients provide (written in C++). One, a *checker* that performs storage system operations and checks that they worked correctly (§3.2). Two, a *storage component* that sets up the checked system (§3.3). Finally, a *checking stack* that combines the first two into a checking harness (§3.4).

3.1 How checked code exposes choice: `choose`

Like prior model checkers [13, 30], EXPLODE provides a function, `choose`, that clients use to select among possible choices in checked code. Given a program point that has `N` possible actions clients insert a call "`choose(N)`," which will appear to fork execution `N` times, returning the values $0, 1, ..., N-1$ in each child execution respectively. They then write code that uses this return value to pick one unique action out of the `N` possibilities. EXPLODE can exhaust all possible actions at this `choose` call by running all forked children. We define a code location that can pick one of several different legal actions to be a *choice point* and the act of doing so a *choice*.

An example: in low memory situations the Linux `kmalloc` function can return `NULL` when called without the `__GFP_NOFAIL` flag. But it rarely does so in practice, making it difficult to comprehensively check that callers correctly handle this case. We can use `choose` to systematically explore both success and failure cases of each `kmalloc` call as follows:

```
void * kmalloc(size_t size, int flags) {
    if((flags & __GFP_NOFAIL) == 0)
        if(choose(2) == 0)
            return NULL;
    ...
```

Typically clients add a small number of such calls. On Linux, we used `choose` to fail six kernel functions: `kmalloc` (as above), `page_alloc` (page allocator), `access_ok` (verify user-provided pointers), `bread` (read a block), `read_cache_page` (read a page), and `end_request` (indicate that a disk request completed). The inserted code mirrors that in `kmalloc`: a call `choose(2)` and an if-statement to pick whether to either (0) return an error or (1) run normally.

3.2 Driving checked code: The checker

The client provides a checker that EXPLODE uses to drive and check a given storage system. The checker implements five methods:

1. `mutate`: performs system-specific operations and calls into EXPLODE to explore choices and to do crash checking.
2. `check`: called after each EXPLODE-simulated crash to check for storage system errors.
3. `get_sig`: an optional method which returns a byte-array signature representing the current state of the checked system. It uses domain-specific knowledge to discard irrelevant details so that EXPLODE knows when two superficially different states are equivalent and avoids repeatedly checking them. The default `get_sig` simply records all choices made to produce the current state.
4. `init` and `finish`: optional methods to set up and clear the checker's internal state, called when EXPLODE mounts and unmounts the checked system.

```
1 : const char *dir = "/mnt/sbd0/test-dir";
2 : const char *file = "/mnt/sbd0/test-file";
3 : static void do_fsync(const char *fn) {
4 :    int fd = open(fn, O_RDONLY);
5 :    fsync(fd);
6 :    close(fd);
7 : }
8 : void FsChecker::mutate(void) {
9 :    switch(choose(4)) {
10:    case 0: systemf("mkdir %s%d", dir, choose(5)); break;
11:    case 1: systemf("rmdir %s%d", dir, choose(5)); break;
12:    case 2: systemf("rm %s", file); break;
13:    case 3: systemf("echo \"test\" > %s", file);
14:       if(choose(2) == 0)
15:          sync();
16:       else {
17:          do_fsync(file);
18:          // fsync parent to commit the new directory entry
19:          do_fsync("/mnt/sbd0");
20:       }
21:       check_crash_now(); // invokes check() for each crash
22:       break;
23:    }
24: }
25: void FsChecker::check(void) {
26:    ifstream in(file);
27:    if(!in)
28:       error("fs", "file gone!");
29:    char buf[1024];
30:    in.read(buf, sizeof buf);
31:    in.close();
32:    if(strncmp(buf, "test", 4) != 0)
33:       error("fs", "wrong file contents!");
34: }
```

Figure 2: Example file system checker. We omit the class initialization code and some sanity checks.

Checkers range from aggressively system-specific (or even code-version specific) to the fairly generic. Their size scales with the complexity of the invariants checked, from a few tens to many thousands of lines.

Figure 2 shows a file system checker that checks a simple correctness property: a file that has been synchronously written to disk (using either the `fsync` or `sync` system calls) should persist after a crash. Mail servers, databases and other application storage systems depend on this behavior to prevent crash-caused data obliteration. While simple, the checker illustrates common features of many checkers, including the fact that it catches some interesting bugs.

The `mutate` method calls `choose(4)` (line 9) to fork and do each of four possible actions: (1) create a directory, (2) delete it, (3) create a test file, or (4) delete it. The first two actions then call `choose(5)` and create or delete one of five directories (the directory name is based on `choose`'s return value). The file creation action calls `choose(2)` (line 14) and forces the test file to disk using `sync` in one child and `fsync` in the other. As Figure 3 shows, one `mutate` call creates thirteen children.

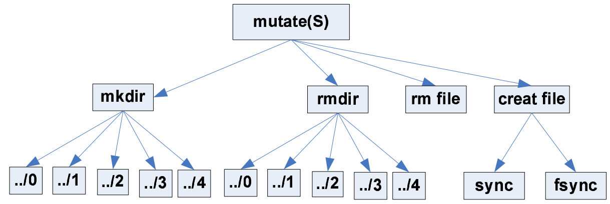


Figure 3: Choices made by one invocation of the `mutate` method in Figure 2's checker. It creates thirteen children.

The checker calls EXPLODE to check crashes. While other code in the system can also initiate such checking, typically it is the `mutate` method's responsibility: it issues operations that change the storage system, so it knows the correct system state and when this state changes. In our example, after `mutate` forces the file to disk it calls the EXPLODE routine `check_crash_now()`. EXPLODE then generates all crash disks at the exact moment of the call and invokes the `check` method on each after repairing and mounting it using the underlying storage component (see § 3.3). The `check` method checks if the test file exists (line 27) and has the right contents (line 32). While simple, this exact checker catches an interesting bug in JFS where upon crash, an `fsync`'d file loses all its contents triggered by the corner-case reuse of a directory inode as a file inode (§7.3 discusses a more sophisticated version of this checker).

So far we have described how a single `mutate` call works. The next section shows how it fits in the checking process. In addition, checking crashes at only a single code point is crude; Section 6 describes the routines EXPLODE provides for more comprehensive checking.

3.3 Setting up checked code: Storage components

Since EXPLODE checks live storage systems, these systems must be up and running. For each storage subsystem involved in checking, clients provide a storage component that implements five methods:

1. `init`: one-time initialization, such as formatting a file system partition or creating a fresh database.
2. `mount`: set up the storage system so that operations can be performed on it.
3. `unmount`: tear down the storage system; used by EXPLODE to clear the storage system's state so it can explore a different one (§5.2).
4. `recover`: repair the storage system after an EXPLODE-simulated crash.
5. `threads`: return the thread IDs for the storage system's kernel threads. EXPLODE reduces nondeterminism by only running these threads when it wants to (§5.2).

```
void Ext3::init(void) {
   // create an empty ext3 FS with user-specified block size
   systemf("mkfs.ext3 -F -j -b %d %s",
     get_option(blk_size), children[0]->path());
}
void Ext3::recover() {
  systemf("fsck.ext3 -y %s", children[0]->path());
}
void Ext3::mount(void) {
   int ret = systemf("sudo mount -t ext3 %s %s",
       children[0]->path(), path());
   if(ret < 0) error("Corrupt FS: Can't mount!");
}
void Ext3::umount(void) {
  systemf("sudo umount %s", path());
}
void Ext3::threads(threads_t &thids) {
   int thid;
   if((thid=get_pid("kjournald")) != -1)
       thids.push_back(thid);
   else
       explode_panic("can't get kjournald pid!");
}
```

Figure 4: Example storage component for the `ext3` file system. The C++ class member `children` chains all storage components that a component is based on; `ext3` has only one child.

Clients write a component once for a given storage system and then reuse it in different checkers. Storage systems tend to be easy to set up, otherwise they will not get used. Thus, components tend to be simple and small since they can merely wrap up already-present system commands (e.g., shell script invocations).

Figure 4 shows a storage component for the `ext3` file system that illustrates these points. Its first four methods call standard `ext3` commands. The one possibly non-obvious method is `threads`, which returns the thread ID of `ext3`'s kernel thread (`kjournald`) using the expedient hack of calling the built-in EXPLODE routine `get_pid` which automatically extracts this ID from the output of the `ps` command.

3.4 Putting it all together: The checking stack

The checking stack builds a checker by glueing storage system components together and then attaching a single checker on top of them. The lowest component of a checking stack typically is a custom RAM disk (downloaded from [24] and slightly modified). While EXPLODE runs on real disks, using a RAM disk avoids non-deterministic interrupts and gives EXPLODE precise, fast control over the contents of a checked system's "persistent" storage. The simplest storage stack attaches a checker to one EXPLODE RAM disk. Such a stack does no useful crash checking, so clients typically glue one or more storage subsystems between these two. Currently a stack can only have one checker. However, there can be a fan-out of storage components, such as setting up multiple RAM disks to make a RAID array. Given a stack, EXPLODE initializes the checked storage stack by calling each `init` bottom up, and then `mount` bottom up. After a crash, it calls the `recover` methods bottom up as well. To unmount, EXPLODE applies `unmount` top down. Figure 5 shows a three-layer storage stack.

```
// Assemble FS + RAID storage stack step by step.
void assemble(Component *&top, TestDriver *&driver) {
  // 1. load two RAM disks with size specified by user
  ekm_load_rdd(2, get_option(rdd, sectors));
  Disk *d1 = new Disk("/dev/rdd0");
  Disk *d2 = new Disk("/dev/rdd1");

  // 2. plug a mirrored RAID array onto the two RAM disks.
  Raid *raid = new Raid("/dev/md0", "raid1");
  raid->plug_child(d1);
  raid->plug_child(d2);

  // 3. plug an ext3 system onto RAID
  Ext3 *ext3 = new Ext3("/mnt/sbd0");
  ext3->plug_child(raid);
  top = ext3; // let eXplode know the top of storage stack

  // 4. attach a file system test driver onto ext3 layer
  driver = new FsChecker(ext3);
}
```

Figure 5: Checking stack: file system checker (Figure 2) on an ext3 file system (Figure 4) on a mirrored RAID array on two EXPLODE RAM disks. We elide the trivial class definitions `Raid` and `Disk`.

4 Implementation Overview

This section gives an overview of EXPLODE. The next two sections discuss the implementation of its most important features: choice and crash checking.

The reader should keep in mind that conceptually what EXPLODE does is very simple. If we assume infinite resources and ignore some details, the following would approximate its implementation:

1. Create a clean initial state (§3.3) and invoke the client's `mutate` on it.
2. At every `choose(N)` call, fork `N` children.
3. On client request, generate all crash disks and run the client `check` method on them.
4. When `mutate` returns, re-invoke it.

This is it. The bulk of EXPLODE is code for approximating this loop with finite resources, mainly the machinery to save and restore the checked system so it can run one child at a time rather than an exponentially increasing number all-at-once. As a result, EXPLODE unsurprisingly looks like a primitive operating system: it has a queue of saved processes, a scheduler that picks which of these jobs to run, and time slices (that start when `mutate` is invoked and end when it returns). EXPLODE's scheduling algorithm: exhaust all possible combinations of choices within a single `mutate` call be-

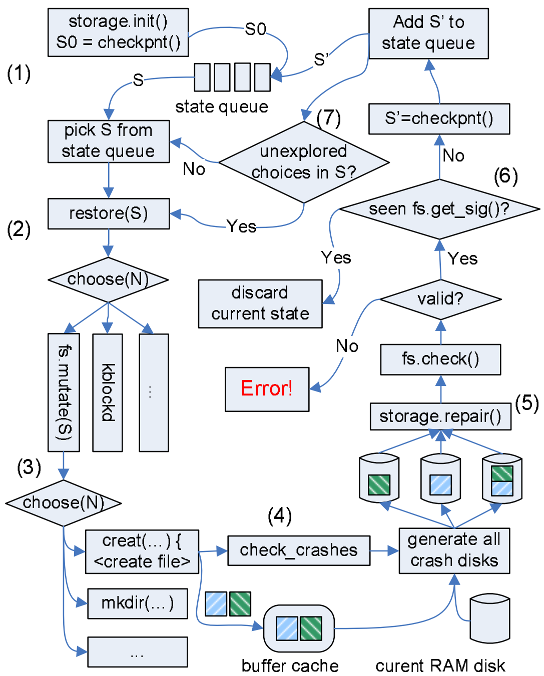


Figure 6: Simplified view of EXPLODE's state exploration loop for the file system checker in Figure 2; some `choose` transitions and method calls elided for space.

fore doing another (§ 2). (Note that turning EXPLODE into a random testing framework is easy: never save and restore states and make each `choose(N)` call return a random integer $[0, N)$ rather than forking, recording each choice for error replay.) The above sketch glosses over some important details; we give a more accurate description below, but the reader should keep this helpful, simplistic one in mind.

From a formal method's perspective, the core of EXPLODE is a simple, standard model checking loop based on exhausting state choices. Figure 6 shows this view of EXPLODE as applied to the file system checker of the previous section; the numbered labels in the figure correspond to the numbers in the list below:

1. EXPLODE initializes the checked system using client-provided `init` methods. It seeds the checking process by saving this state and putting it on the state queue, which holds all states (jobs) to explore. It separately saves the created disk image for use as a pristine initial disk.
2. The EXPLODE "scheduler" selects a state `S` from its state queue, restores it to produce a running storage system, and invokes `choose` to run either the `mutate` method or one of the checked systems' kernel threads. In the figure, `mutate` is selected.
3. `mutate` invokes `choose` to pick an action. In our example it picks `creat` and calls it, transferring

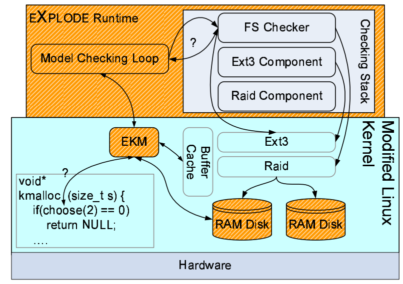


Figure 7: Snapshot: EXPLODE with Figure 5's checking stack

control to the running Linux kernel. The `creat` system call writes two dirty blocks to the buffer cache and returns back to `mutate`.

4. `mutate` calls EXPLODE to check that the file system correctly recovers from any crash at this point.
5. EXPLODE generates combinations of disks that could be seen after a crash. It then runs the client code to: `mount` the crash disk, `recover` it, and `check` it. If these methods flag an error or they crash, EXPLODE records enough information to recreate this error, and stops exploring this state.
6. Otherwise EXPLODE returns back into `mutate` which in turn returns. EXPLODE checks if it has already seen the current state using the abstracted representation returned by `get_sig`. If it has, it discards the state to avoid redundant work, otherwise it checkpoints it and puts it on the state queue.
7. EXPLODE then continues exploring any remaining choices in the original state `S`. If it has exhausted all choice combinations on `S` it picks a previously saved state off the state queue and repeats this process on it. This loop terminates when the state queue is empty or the user loses patience. (The number of possible states means the former never happens.)

After crash checking, the checked system may have a butchered internal state. Thus, before continuing, EXPLODE restores a clean copy of the current state without doing crash checking (not pictured). In addition, since checking all possible crash disks can take too long, users can set a deterministic threshold: if the number of crash disks is bigger than this threshold, EXPLODE checks a configurable number of random combinations.

Figure 7 gives a snapshot of EXPLODE; Table 1 breaks down the lines of code for each of the components. It consists of two user-level pieces: a client-provided checking stack and the EXPLODE runtime, which implements most of the model checking loop described above. EXPLODE also has three kernel-level pieces: (1) one or more RAM disks, (2) a custom kernel module, EKM, and (3) a modified Linux kernel (either version 2.6.11 or 2.6.15). EXPLODE uses EKM to monitor and determinis-

	Name	Line Count
Linux	**EKM**	1,261
	RAM disk Driver	326
	Kernel Patch	328
	EKM-generated	2,194
BSD	**EKM**	729
	RAM disk Driver	357
	Kernel Patch	116
User-mode	**EXPLODE**	5,802
	RPC Library	521

Table 1: EXPLODE lines of code (ignoring comments and blank lines), broken down by modules. The EKM driver contains 2,194 lines of automatically generated code (**EKM-generated**). The EXPLODE runtime and the RPC library run at user-level, the rest is in the kernel. The RPC library is used to check virtual machines (§ 9.3). **BSD** counts are smaller because this port does not yet provide all EXPLODE features.

tically control checking-relevant actions done by kernel code and record system events needed for crashes. The modified kernel calls EKM to log system events and when it reaches a choice point. These modifications add 328 lines of mostly read-only instrumentation code, typically at function entry or exit. We expect them to generally be done by EXPLODE users. Unlike EXPLODE's user-space code, its RAM disk driver and EKM are kernel-specific, but are fairly small and easily ported to a new OS. We recently ported EXPLODE's core to FreeBSD, which Section 10 describes in more detail.

Given all of these pieces, checking works as follows. First, the user compiles and links their code against the EXPLODE runtime, and runs the resultant executable. Second, the EXPLODE runtime dynamically loads its kernel-level components and then initializes the storage system. Finally, EXPLODE explores the checked system's states using its model checking loop.

While checking a live kernel simplifies many things, the downside is that many bugs we find with EXPLODE cause kernel crashes. Thus, we run the checked system inside a virtual machine monitor (VMware Workstation), where it can blow itself up without hurting anyone. This approach also makes checking a non-super-user operation, with the usual benefits.

5 Exploring Choices

EXPLODE exhausts a choice point by checkpointing the current state `S`, exploring one choice, restoring `S`, and then exploring the other choices. Below we discuss how EXPLODE implements checkpoint and restore by replaying choices (§ 5.1) deterministically (§ 5.2).

5.1 Checkpointing and restoring states.

A standard checkpoint implementation would copy the current system state to a temporary buffer, which restore would then copy back. Our previous storage checking system, FiSC, did just this [30]. Unfortunately, one cannot simply save and restore a kernel running on raw hardware, so we had to instead run a heavily-hacked Linux kernel inside FiSC at user level, turning FiSC into a primitive virtual machine. Doing so was the single largest source of FiSC complexity, overhead to check new systems, and limitation on what we could check.

EXPLODE uses computation rather than copying to recreate states. It checkpoints a state `S` by recording the set of choices the checked code took to reach `S`. It restores `S` by starting from a clean initial state and replaying these choices. Thus, assuming deterministic actions, this method regenerates `S`. Mechanically, checkpoint records the sequence of n choices that produced `S` in an array; during replay the ith `choose` call simply returns the ith entry in this array.

This one change led to orders of magnitude reduction in complexity and effort in using EXPLODE as opposed to FiSC, to the degree that EXPLODE completely subsumes our prior work in almost every aspect by a large amount. It also has the secondary benefit that states have a tiny representation: a sequence of integers, one for each choice point, where the integer specifies which of N choices were made. Note that some model checkers (and systems in other contexts [10]) already use replay-recreation of states, but for error reporting and state size reduction, rather than for reducing invasiveness. One problem with the approach is that the restored state's timestamps will not match the original, making it harder to check some time properties.

Naively, it might seem that to reset the checked systems' state we have to reboot the machine, re-initialize the storage system, mount it, and only then replay choices. This expensive approach works, but fortunately, storage systems have the observed, nice property that simply unmounting them clears their in-memory state, removing their buffer cache entries, freeing up their kernel data structures, etc. Thus, EXPLODE uses a faster method: call the client-supplied `unmount` to clear the current state, then load a pristine initial state (saved after initialization) using the client-supplied `mount`.

It gets more costly to restore states as the length of their choice sequence grows. Users can configure EXPLODE to periodically chop off the prefix of choice sequences. It does so by (1) calling `unmount` to force the checked system state to disk and (2) using the resultant disk image as a new initial state that duplicates the effect of the choices before the `unmount` call. The downside is that it can no longer reorder buffer cache entries from before this point during crash checking.

5.2 Re-executing code deterministically

EXPLODE's restore method only works if it can deterministically replay checked code. We discuss how EXPLODE does so below, including the restrictions imposed on the checked system.

Doing the same choices. Kernel code containing a `choose` call can be invoked by non-checking code, such as interrupt handlers or system calls run by other processes. Including such calls makes it impossible to replay traces. EXPLODE filters them by discarding any calls from an interrupt context or calls from any process whose ID is not associated with the checked system.

Controlling threads. EXPLODE uses priorities to control when storage system threads run (§ 4, bullet 2). It quiesces storage system threads by giving them the lowest priority possible using an EKM `ioctl`. It runs a thread by giving it a high priority (others still have the lowest) and calling the kernel scheduler, letting it schedule the right thread. It might seem more sensible for EXPLODE to orchestrate thread schedules via semaphores. However, doing so requires intrusive changes and, in our experience [30], backfires with unexpected deadlock since semaphores prevent a given thread from running even if it absolutely must. Unfortunately, using priorities is not perfect either, and still allows non-deterministic thread interleaving. We detect pathological cases where a chosen thread does not run, or other "disabled" threads do run using the "last-run" timestamps in the Linux process data structure. These sanity checks let us catch when we generate an error trace that would not be replayable or when replaying it takes a different path. Neither happens much in practice.

Requirements on the checked system. The checked system must issue the same `choose` calls across replay runs. However, many environmental features can change across runs, providing many sources of potential non-deterministic input: thread stacks in different locations, memory allocations that return different blocks, data structures that have different sizes, etc. None of these perturbations should cause the checked code to behave differently. Fortunately, the systems we checked satisfy this requirement "out of the box" — in part because they are isolated during checking, and nothing besides the checker and their kernel threads call into them to modify their RAM disk(s). Non-deterministic systems require modification before EXPLODE can reliably check them. However, we expect such cases to rarely occur. If nothing else, usability forces systems to ensure that re-executing the same user commands produces the same system state. As a side-effect, they largely run the same code paths (and thus would hit the same `choose` calls).

While checked code must do the same `choose` calls for deterministic error replay, it does not have to allocate the same physical blocks. EXPLODE replays choices, but then regenerates all different crash combinations after the last choice point until it (re)finds one that fails checking. Thus, the checked code can put logical contents in different physical blocks (e.g., an inode resides in disk block 10 on one run and in block 20 on another) as long as the logical blocks needed to cause the error are still marked as dirty in the buffer cache.

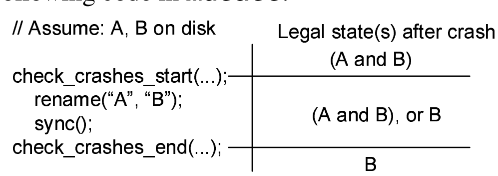
6 Checking Crashes

This section discusses crash checking issues: EXPLODE's checking interface (§ 6.1), how it generates crash disks (§ 6.2), how it checks crashes during recovery (§ 6.3), how it checks for errors caused by application crashes (§ 6.4), and some refinements (§ 6.5).

6.1 The full crash check interface

The `check_crashes_now()` routine is the simplest way to check crashes. EXPLODE also provides a more powerful (but complex) interface clients can use to directly inspect the log EXPLODE extracts from EKM. They can also add custom log records. Clients use the log to determine what state the checked system should recover to. They can initiate crash checking at any time while examining the log. For space reasons we do not discuss this interface further, though many of our checkers use it. Instead we focus on two simpler routines `check_crashes_start` and `check_crashes_end` that give most of the power of the logging approach.

Clients call `check_crashes_start` before invoking the storage system operations they want to check and `check_crashes_end` after. For example, assume we want to check if we can atomically rename a file `A` to `B` by calling `rename` and then `sync()`. We could write the following code in `mutate`:



EXPLODE generates all crash disks that can occur (inclusively) between these calls, invoking the client's `check` method on each. Note how the state the system should recover to changes. At the `check_crashes_start` call, the recovered file system should contain both `A` and `B`. During the process of renaming, the recovered file system can contain either (1) the original `A` and `B` or (2) `B` with `A`'s original contents. After `sync` completes, only `B` with `A`'s original contents should exist.

This pattern of having an initial state, a set of legal intermediate states, and a final state is a common one for checking. Thus, EXPLODE makes it easy for `check` to distinguish between these epochs by passing a flag that tells `check` if the crash disk could occur at the first call (`EXP_BEGIN`), the last call (`EXP_END`), or in between (`EXP_INBETWEEN`). We could write a check method to use these flags as follows:

```
check(int epoch, ...) {
    if(epoch == EXP_BEGIN)
        // check (A and B)
    else if(epoch == EXP_INBETWEEN)
        // check (A and B) or B
    else //    EXP_END
       // check B
}
```

EXPLODE uses C++ tricks so that clients can pass an arbitrary number of arguments to these two routines (up to a user-specified limit) that in turn get passed to their `check` method.

6.2 Generating crash disks

EXPLODE generates crash disks by first constructing the current *write set*: the set of disk blocks that currently *could be* written to disk. Linux has over ten functions that affect whether a block can be written or not. The following two representative examples cause EXPLODE to add blocks to the write set:

1. `mark_buffer_dirty(b)` sets the dirty flag of a block `b` in the buffer cache, making it eligible for asynchronous write back.
2. `generic_make_request(req)` submits a list of sectors to the disk queue. EXPLODE adds these sectors to the write set, even if they are clean, which can happen for storage systems maintaining their own private buffer caches (as in the Linux port of XFS).

The following three representative examples cause EXPLODE to remove blocks from the write set:

1. `clear_buffer_dirty(b)` clears `b`'s dirty flag. The buffer cache does not write clean buffers to disk.
2. `end_request()`, called when a disk request completes. EXPLODE removes all versions of the request's sectors from the write set since they are guaranteed to be on disk.
3. `lock_buffer(b)`, locks `b` in memory, preventing it from being written to disk. A subsequent `clear_buffer_locked(b)` will add `b` to the write set if `b` is dirty.

Writing any subset of the current write set onto the current disk contents generates a disk that could be seen if the system crashed at this moment. Figure 8 shows how EXPLODE generates crash disks; its numbered labels correspond to those below:

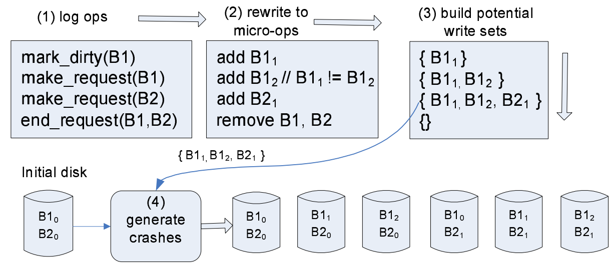


Figure 8: Generating all potential crash disks.

1. As the storage system executes, EKM logs operations that affect which blocks could be written to disk.
2. EXPLODE extracts this log using an EKM `ioctl` and reduces the logged operations to micro-operations that add or remove blocks from the write set.
3. It then applies these add and remove operations, in order, to the initial write set.
4. Whenever the write set shrinks, it generates all possible crash disks by applying all subsets of the write set to the current disk. (Doing so when the write set shrinks rather than grows makes it trivial to avoid duplicate work.)

Note that the write set tracks a block's contents in addition to the block itself. Naively it may appear that when EXPLODE adds a block b to the write set it should replace any previous copy of b with this more recent one. (Our previous work [30] did exactly this.) However, doing so misses errors. For example, in the figure, doing so misses one crash disk $(B1_1, B2_1)$ since the second insertion of block $B1$ replaces the previous version $B1_1$ with $B1_2$.

6.3 Checking crashes during recovery

Clients can also use EXPLODE to check that storage systems correctly handle crashes during recovery. Since environmental failures are correlated, once one crash happens, another is not uncommon: power may flicker repeatedly in a storm or a machine may keep rebooting because of a bad memory board. EXPLODE generates the disks that could occur if recovery crashes, by tracking the write set produced while running `recover`, and then applying all its subsets to the initial crash disk. It checks these "crash-crash" disks as it would a crash disk. Note this assumes recovery is idempotent in that if a correct recovery with no crash produces state S_{valid} then so should a prematurely crashed repair followed by a successful one. We do not (but could) check for further crashes during recovery since implementors seem uninterested in such errors [30].

6.4 Checking "soft" application crashes

In addition to "hard" machine crashes that wipe volatile state, EXPLODE can also check that applications correctly recover from "soft" crashes where they crashed, but the operating system did not. Such soft crashes are usually more frequent than hard crashes with causes ranging from application bugs to impatient users pressing "ctrl-C." Even applications that ignore hard crashes should not corrupt user data because of a soft crash.

EXPLODE checks soft crashes in two steps. First, it runs the checker's `mutate` method and logs all mutating file system operations it performs. Second, for each log prefix EXPLODE mounts the initial disk and replays the operations in the prefix in the order they are issued. If the log has n operations EXPLODE generates n storage states, and passes each to the `check` method.

6.5 Refinements

In some cases we remove blocks from the write set too eagerly. For example, we always remove the sectors associated with `end_request`, but doing so can miss permutations since subsequent writes may not in fact have waited for (depended on) the write to complete. Consider the events: (1) a file system writes sector S1, (2) the write completes, (3) it then writes sector S2. If the file system wrote S2 without explicitly waiting for the S1 write to complete then these writes could have been reordered (i.e., there is no happens-before dependency between them). However, we do not want to grovel around inside storage systems rooting out these false dependencies, and conservatively treat all writes that complete as waited for. A real storage system implementor could obviously do a better job.

To prevent the kernel from removing buffers from the write set, we completely disable the dirty buffer flushing threads `pdflush`, and only schedule the kernel thread `kblockd` that periodically flushes the disk queue between calls to the client `mutate` method.

If a checked system uses a private buffer cache, EXPLODE cannot see all dirty blocks. We partially counter this problem by doing an unmount before generating crash disks, which will flush all private dirty buffers to disk (when EXPLODE can add them to its write set). Unfortunately, this approach is not a complete solution since these unmount-driven flushes can introduce spurious dependencies (as we discussed above).

7 In-Depth Checking: File Systems

This section demonstrates that EXPLODE's lightweight approach does not compromise its power by replicating (and sometimes superseding) the results we obtained with our previous, more strenuous approach [30]. It also shows EXPLODE's breadth by using it to check ten Linux file systems with little incremental effort.

We applied EXPLODE to all but one of the disk based file systems on Linux 2.6.11: ext2, ext3, JFS, ReiserFS, Reiser4, XFS, MSDOS, VFAT, HFS, and HFS+. We skipped NTFS because repairing a crashed NTFS disk requires mounting it in Windows. For most file systems, we used the most up-to-date utilities in the Debian "etch" Linux distribution. For HFS and HFS+, we had to download the source of their utilities from OpenDarwin [14] and compile it ourselves. The storage components for these file systems mirror `ext3`'s component (§ 3.3). Four file systems use kernel threads: JFS, ReiserFS, Reiser4 and XFS. We extracted these thread IDs using the same trick as with `ext3`.

While these file systems vary widely in terms of implementation, they are identical in one way: none give clean, precise guarantees of the state they recover to after a crash. As a result, we wrote three checkers that focused on different special cases where what they did was somewhat well-defined. We built these checkers by extending a common core, which we describe below. We then describe the checkers and the bugs they found.

7.1 The generic checker core

The basic checker starts from an empty file system and systematically generates file system topologies up to a user-specified number of files and directories. Its `mutate` exhaustively applies each of the following eight system calls to each node (file, link, directory) in the current topology before exploring the next: `ftruncate`, `pwrite` (which writes to a given offset within a file), `creat`, `mkdir`, `unlink`, `rmdir`, `link` and `rename`. For example, if there are two leaf directories, the checker will delete both, create files in both, etc. Thus, the number of possible choices for a given tree grows (deterministically) with its size. For file systems that support holes, the checker writes at large offsets to exercise indirect blocks. Other operations can easily be added.

For each operation it invokes, `mutate` duplicates its effect on a fake "abstract" file system it maintains privately. For example, if it performs three operations `mkdir(/a)`, `mkdir(/a/b)`, and `sync()` then the abstract file system will be the tree `/a/b`, which the real file system must match exactly. The checker's `get_sig` method returns a canonical version of this abstract file system. This canonicalization mirrors that in [30], and uses relabeling to make topologies differing only in naming equivalent and discards less interesting properties such as timestamps, actual disk blocks used, etc.

7.2 Check: Failed system calls have no effect

This check does not involve crash-recovery. It checks that if a file system operation (except `pwrite`) returns an error, the operation has no user-visible effect. It uses

EXPLODE to systematically fail calls to the six kernel functions discussed in Section 3.1. The actual check uses the abstract file system described in the previous subsection. If a system call succeeds, the checker updates the abstract file system, but otherwise does not. It then checks that the real file system matches the abstract one.

Bugs found. We found 2 bugs in total. One of them was an unfixed Linux VFS bug we already reported in [30]. The other one was a minor bug in ReiserFS `ftruncate` which can fail with its job half-done if memory allocation fails. We also found that Reiser4 calls `panic` on memory allocation failures, and ReiserFS calls `panic` on disk read failures. (We did not include these two undesired behaviors in our bug counts.)

7.3 Check: "sync" operations work

Applications such as databases and mail servers use operating system-provided methods to force their data to disk in order to prevent crashes from destroying or corrupting it. Unfortunately, they are completely at these routines' mercy — there is no way to check they do what they claim, yet their bugs can be almost arbitrarily bad.

Fortunately, EXPLODE makes it easy to check these operations. We built a checker (similar to the one in Figure 2) to check four methods that force data to disk:

1. `sync` forces all dirty buffers to disk.
2. `fsync(fd)` forces `fd`'s dirty buffers to disk.
3. Synchronously mounted file system: a system call's modifications are on disk when the call returns.
4. Files opened with `O_SYNC`: all modifications done by a system call through the returned file descriptor are on disk when the call returns.

After each operation completes and its modifications have been forced to disk, the sync-checker tells EXPLODE to do crash checking and verifies that the modifications persist.

Note, neither `fsync` nor `O_SYNC` guarantee that directory entries pointing to the sync'd file are on disk, doing so requires calling `fsync` on any directory containing the file (a legal operation in Linux). Thus, the checker does an `fsync` on each directory along the path to the sync'd file, ensuring there is a valid path to it in the recovered file system.

Bugs found. Table 2 summarizes the 13 bugs found with this checker. Three bugs show up in multiple ways (but are only counted three times): a VFS limitation caused all file systems to fail the `O_SYNC` check, and both HFS and HFS+ mangled file and directory permissions after crashing, therefore failing all four sync checks. We describe a few of the more interesting bugs below.

Besides HFS/HFS+, both MSDOS and VFAT mishandled `sync`. Simple crashes after `sync` can introduce directory loops. The maintainers confirmed they knew of these bugs, though they had not been publicly disclosed. These bugs have subsequently been fixed. Eight file systems had synchronous mount bugs. For example, `ext2` gives no consistency guarantees by default, but mounting it synchronously still allows data loss.

FS	`sync`	`mount sync`	`fsync`	`O_SYNC`
ext2		✘	✘	✘
ext3				✘
ReiserFS		✘		✘
Reiser4				✘
JFS		✘	✘	✘
XFS		✘		✘
MSDOS	✘	✘		✘
VFAT	✘	✘		✘
HFS	✘	✘	✘	✘
HFS+	✘	✘	✘	✘

Table 2: Sync checking results: ✘ indicates the file system failed the check. There were 13 bugs, three of which show up more than once, causing more ✘ marks than errors.

There were two interesting `fsync` errors, one in JFS (§3.2) and one in `ext2`. The `ext2` bug is a case where an implementation error points out a deeper design problem. The bug occurs when we: (1) shrink a file "A" with `truncate` and (2) subsequently `creat`, `write`, and `fsync` a second file "B." If file B reuses the indirect blocks of A freed via `truncate`, then following a crash `e2fsck` notices that A's indirect blocks are corrupt and clears them, destroying the contents of B. (For good measure it then notices that A and B share blocks and "repairs" B by duplicating blocks from A.) Because ext2 makes no guarantees about what is written to disk, fundamentally one cannot use `fsync` to *safely* force a file to disk, since the file can still have implicit dependencies on other file system state (in our case if it reuses an indirect blocks for a file whose inode has been cleared in memory but not on disk).

7.4 Check: a recovered FS is "reasonable"

Our final check is the most stringent: after a crash a file system recovers to a "reasonable" state. No files, directories, or links flushed to disk are corrupted or disappear (unless explicitly deleted). Nor do they spontaneously appear without being created. For example, if we crash after performing two operations `mkdir(/A)` and `mkdir(/A/B)` on an empty file system, then there are exactly three correct recovered file systems: (1) / (no data), (2) `/A`, or (3) `/A/B`. We should not see directories or files we never created. Similarly, if `/A` was forced to disk before the crash, it should still exist.

For space reasons we only give a cursory implementation overview. As `mutate` issues operations, it builds two sets: (1) the stable set, which contains the operations it knows are on the disk, (2) the volatile set, which contains the operations that may or may not be on disk. The `check` method verifies that the recovered file system can be constructed using some sequence of volatile operations legally combined with all the stable ones. The implementation makes heavy use of caching to prune the search and "desugars" operations such as `mkdir` into smaller atomic operations (in this case it creates an inode and then forms a link to it) to ensure it can describe their intermediate effects.

Bugs found. We applied this check to ext2, ext3, JFS, ReiserFS and Reiser4. Unsurprisingly, since `ext2` gives no crash guarantees, files can point to uninitialized blocks, and sync'd files and directories can be removed by its `fsck`. Since JFS journals metadata but not data, its files can also point to garbage. These behaviors are design decisions so we did not include them in our bug counts. We found two bugs (one in JFS, one in Reiser4) where crashed disks cannot be recovered by `fsck`. We could not check many topologies for ReiserFS and Reiser4 because they appear to leak large amounts of memory on every `mount` and `unmount` (Our bug counts do not include these leaks.)

In addition, we used the crash-during-recovery check (§6.3) on Reiser4. It found a bug where Reiser4 becomes so corrupted that mounting it causes a kernel panic. (Since our prior work explored this check in detail we did not apply it to more systems.)

Finally, we did a crude benchmark run by running the checker (without crash-during-recovery checking) to `ext3` inside a virtual machine with 1G memory on a Intel P4 3.2GHZ with 2G memory. After about 20 hours, EXPLODE checked 230,744 crashes for 327 different FS topologies and 1582 different FS operations. The run died because Linux leaks memory on each `mount` and `unmount` and runs out of memory. Although we fixed two leaks, more remain (we did not count these obliquely-detected errors in our bug counts but were tempted to). We intend to have EXPLODE periodically checkpoint itself so we can reboot the machine and let EXPLODE resume from the checkpoints.

8 Even Simple Checkers Find Bugs

This section shows that even simple checkers find interesting bugs by applying it to three version control systems and the Berkeley DB database.

The next two sections demonstrate that EXPLODE works on many different storage systems by applying it to many different ones. The algorithm for this process: write a quick checker, use it to find a few errors, declare success, and then go after another storage system. In all cases we could check many more invariants. Table 3 summarizes all results.

System	Storage	Checker	Bugs
FS	744	5,477	18
CVS	27	68	1
Subversion	-	-	1
EXPENSIV	30	124	3
Berkeley DB	82	202	6
RAID	144	FS + 137	2
NFS	34	FS	4
VMware GSX/Linux	54	FS	1
Total	1,115	6,008	36

Table 3: Summary of all storage systems checked. All line counts ignore comments and whitespace. **Storage** gives the line count for each system's storage component, which for **FS** includes the components for all ten file systems. **Checker** gives the checker line counts, which for EXPENSIV includes two checkers. We reused the FS checker to check RAID, NFS and VMware. We wrote an additional checker for RAID. We checked Subversion using an early version of EXPLODE; we have not yet ported its component and checker.

8.1 Version control software

This section checks three version control systems: CVS, Subversion [27], and an expensive commercial system we did not have source code for, denoted as EXPENSIV (its license precludes naming it directly). We check that these systems meet their fundamental goal: do not lose or corrupt a committed file. We found errors in all three.

The storage component for each wraps up the commands needed to set up a new repository on top of one of the file systems we check. The checker's `mutate` method checks out a copy of the repository, modifies it, and commits the changes back to the main repository. After this commit completes, these changes should persist after any crash. To test this, `mutate` immediately calls `check_crashes_now()` after the commit completes. The `check` method flags an error if: (1) the version control systems' crash recovery tool (if any) gives an error or (2) committed files are missing.

Bugs found. All three systems made the same mistake. To update a repository file `A` without corrupting it, they first update a temporary file `B`, which they then atomically rename to `A`. However, they forget to force `B`'s contents to disk before the rename, which means a crash can destroy it.

In addition EXPENSIV purports to atomically merge two repositories into one, where any interruption (such as crash) will either leave the two original repositories or one entirely (correctly) merged one. EXPLODE found a bug where a crash during merge corrupts the repository, which EXPENSIV's recovery tool (`EXPENSIV -r check -f`) cannot fix. This error seems to be caused by the same renaming mistake as above.

Finally, we found that even a soft crash during a merge corrupts EXPENSIV's repository. It appears EXPENSIV renames multiple files at the end of the merge. Although each individual rename is atomic against a soft crash, their aggregation is not. The repository is corrupted if not all files are renamed.

8.2 Berkeley DB

The database checker in this section checks that after a crash no committed transaction records are corrupted or disappear, and no uncommitted ones appear. It found six bugs in Berkeley DB 4.3 [2].

Berkeley DB's storage component only defines the `init` method, which calls Berkeley DB utilities to create a database. It does not require `mount` or `unmount`, and has no threads. It performs recovery when the database is opened with the `DB_RECOVER` flag (in the `check` method). We stack this component on top of a file system one.

The checker's `mutate` method is a simple loop that starts a transaction, adds several records to it, and then commits this transaction. It records committed transactions. It calls `check_crashes_start` before each commit and `check_crashes_end` (§ 6.1) after to verify that there is a one-to-one mapping between the transactions it committed and those in the database.

Bugs found. We checked Berkeley DB on top of ext2, ext3, and JFS. On ext2 creating a database inside a transaction, while supposedly atomic, can lead to a corrupted database if the system crashes before the database is closed or `sync` is manually called. Furthermore, even with an existing database, committed records can disappear during a crash. On ext3 an unfortunate crash while adding a record to an existing database can again leave the database in an unrecoverable state. Finally, on all three file systems, a record that was added but never committed can appear after a crash. We initially suspected these errors came from Berkeley DB incorrectly assuming that file system blocks were written atomically. However, setting Berkeley DB to use sector-aligned writes did not fix the problem. While the errors we find differ depending on the file system and configuration settings, some are probably due to the same underlying problem.

9 Checking "Transparent" Subsystems

Many subsystems transparently slip into a storage stack. Given a checker for the original system, we can easily check the new stack: run the same checker on top of it and make sure it gives the same results.

9.1 Software RAID

We ran two checkers on RAID. The first checks that a RAID transparently extends a storage stack by running the file system sync-checker (§ 7.3) on top of it. A file system's crash and non-crash behavior on top of RAID should be the same as without it: any (new) errors the checker flags are RAID bugs. The second checks that losing any single sector in a RAID1 or RAID5 stripe does not cause data loss [20]. I.e., the disk's contents were always correctly reconstructed from the non-failed disks.

We applied these checks to Linux's software RAID [26] levels 1 and 5. Linux RAID groups a set of disks and presents them as a single block device to the rest of the system. When a block write request is received by the software RAID block device driver, it recomputes the parity block and passes the requests to the underlying disks in the RAID array. Linux RAID repairs a disk using a very simple approach: overwrite all of the disk's contents, rather than just those sectors that need to be fixed. This approach is extremely slow, but also hard to mess up. Still, we found two bugs.

The RAID storage component methods map directly to different options for its administration utility `mdadm`. The `init` method uses `mdadm --create` to assemble either two or four RAM disks into a RAID1 or RAID5 array respectively. The `mount` method calls `mdadm --assemble` on these disks and the `unmount` method tears down the RAID array by invoking `mdadm --stop`. The `recover` method reassembles and recovers the RAID array. We used the `mdadm --add` command to replace failed disks after a disk failure. The checking stack is similar to that in Figure 5.

Bugs found. The checker found that Linux RAID does not reconstruct the contents of an unreadable sector (as it easily could) but instead marks the *entire* disk that contains the bad sector as faulty and removes it from the RAID array. Such a fault-handling policy is not so good: (1) it makes a trivial error enough to prevent the RAID from recovering from *any* additional failure, and (2) as disk capacity increases, the probability that another sector goes bad goes to one.

Given this fault-handling policy, it is unsurprising our checker found that after two sector read errors happen on different disks, requiring manual maintenance, almost all maintenance operations (such as `mdadm --stop` or `mdadm --add`) fail with a "Device or resource busy" error. Disk write requests also fail in this case, rendering the RAID array unusable until the machine is rebooted. One of the main developers confirmed that these behaviors were bad and should be fixed with high priority [4].

9.2 NFS

NFS synchronously forces modifications to disk before requests return [23]. Thus, with only a single client modifying an NFS file system, after a crash NFS must recover to the same file system tree as a local file system mounted synchronously. We check this property by running the sync-checker (§7.3) on NFS and having it treat NFS as a synchronously mounted file system. This check found four bugs when run on the Linux kernel's NFS (NFSv3) implementation [19].

The NFS storage component is a trivial 15-lines of code (plus a hand-edit of "`/etc/exports`" to define an NFS mount point). It provides two methods: (1) `mount`, which sets up an NFS partition by exporting a local FS over the NFS loop-back interface and (2) `unmount`, which tears down an NFS partition by unmounting it. It does not provide a `recover` method since the `recover` of the underlying local file system must be sufficient to repair crashed NFS partitions. We did not model network failures, neither did we control the scheduling of NFS threads, which could make error replay non-deterministic (but did not for ours).

Bugs found. The checker found a bug where a client that writes to a file and then reads the same file through a hard link in a different directory will not see the values of the first write. We elide the detailed cause of this error for space, other than noting that diagnosing this bug as NFS's fault was easy, because it shows up regardless of the underlying file system (we tried ext2, ext3, and JFS).

We found additional bugs specific to individual file systems exported by NFS. When JFS is exported over NFS, the `link` and `unlink` operations are not committed synchronously. When an ext2 file system is exported over NFS, our checker found that many operations were not committed synchronously. If the NFS server crashes these bugs can lose data and cause data values to go backwards for remote clients.

9.3 VMware GSX server

In theory, a virtual machine slipped beneath a guest OS should not change the crash behavior of a correctly-written guest storage system. Roughly speaking, correctness devolves to not lying about when a disk block actually hits a physical disk. In practice, speed concerns make lying tempting. We check that a file system on top of a virtual machine provided "disk" has the same synchronous behavior as running without it (again) using the sync-checker (§7.3). We applied this check to VMware GSX 3.2.0 [29] running on Linux. GSX is an interesting case for EXPLODE: a large, complex commercial system (for which we lack source code) that, from the point of view of a storage system checker, implements a block device interface in a strange way.

The VMware GSX scripting API makes the storage component easy to build. The `init` method copies a precreated empty virtual disk image onto the file system on top of EXPLODE RAM disk. The `mount` method starts the virtual machine using the command

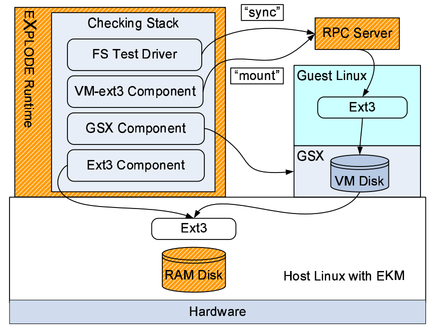


Figure 9: The VMware checking stack.

`vmware-cmd start` and `unmount` stops it using `vmware-cmd stop hard`. The `recover` method calls `vmware-cmd start`, which repairs a crashed virtual machine, and then removes a dangling lock (created by the "crashed" virtual machine to prevent races on the virtual disk file).

As shown in Figure 9 the checking stack was the most intricate of this paper. It has five layers, starting from bottom to top: (1) a RAM disk, (2) the ext3 file system in the host, storing the GSX virtual disk file, (3) GSX, (4) the ext3 file system in the guest, (5) the sync-checker. The main complication in building this stack was the need to split EXPLODE into two pieces, one running in the host, the other in the guest. Since the virtual machine will frequently "crash" we decided to keep the part running inside it simple and make it a stateless RPC server. The entire storage stack and the sync-checker reside in the host. When the sync-checker wants to run an operation in the guest, or a storage method wants to run a utility, they do RPC calls to the server in the guest, which then performs the operation.

Bugs found. Calling `sync` in the guest OS does not correctly flush dirty buffers to disk, but only to the host's buffer cache. According to VMware documents, setting the "disable write caching" configuration flag forces all writes to disk. However, we hit the same bug even with this flag on. This bug makes it impossible to reliably run a storage system on top of this VMM on Linux. We confirmed this problem with one of the main developers who stated that it should not show up in the latest version [28].

10 Checking on a new system: FreeBSD

We ported EXPLODE to FreeBSD 6.0 to ensure porting was easy and to shake out Linux-specific design assumptions. We spent most of our time writing a new RAM disk and EKM module; we only needed to change a few lines in the user-level runtime to run on FreeBSD.

The FreeBSD version of EXPLODE supports crash checking, but currently does not provide a kernel-level `choose` nor logging of system calls. Neither should present a challenge here or in general. Even without these features, we reproduced the errors in CVS and EXPENSIV we saw on Linux as well as finding new errors in FreeBSD UFS2. Below, we discuss issues in writing EKM and the RAM disk.

EKM. Crash checking requires adding calls to EKM in functions that mark buffers as clean, dirty, or write them to disk. While a FreeBSD developer could presumably enumerate all such functions easily, our complete lack of experience with FreeBSD meant it took us about a week to find all corner-cases. For example, FreeBSD's UFS2 file system sometimes bypasses the buffer cache and writes directly to the underlying disk.

There were also minor system-differences we had to correct for. As an example, while Linux and FreeBSD have similar structures for buffers, they differ in how they store bookkeeping information (e.g., representing offsets in sectors on Linux, and in bytes on FreeBSD). We adjusted for such differences inside EKM so that EXPLODE's user-level runtime sees a consistent interface. We believe porting should generally be easy since EKM only logs the offset, size, and data of buffer modifications, as well as the ID of the modifying thread. All of these should be readily available in any OS.

RAM disk. We built our FreeBSD RAM disk by modifying the `/dev/md` memory-based disk device. We expect developers can generally use this approach: take an existing storage device driver and add trivial `ioctl` commands to read and write its disk state by copying between user- and kernel-space.

Bug-Finding Results. In addition to our quick tests to replicate the EXPENSIV and CVS bugs, we also ran our sync-checker (§7.3) on FreeBSD's UFS2 with soft updates disabled. It found errors where `fsck` with the `-p` option could not recover from crashes. While `fsck` without `-p` could repair the disk, the documentation for `fsck` claims `-p` can recover from all errors unless unexpected inconsistencies are introduced by hardware or software failures. Developers confirmed that this is a problem and should be further investigated.

11 Related Work

Below we compare EXPLODE to file system testing, software model checking, and static bug finding.

File system testing tools. There are many file system testing frameworks that use application interfaces to stress a "live" file system with an adversarial environment. These testing frameworks are less comprehensive than our approach, but they work "out of the box." Thus, there is no reason not to both test a file system and then test with EXPLODE (or vice versa).

Recently, Prabhakaran *et al* [21] studied how file systems handle disk failures and corruption. They developed a testing framework that uses techniques from [25] to infer disk block types and then inject "type-aware" block failure and corruption into file systems. Their results provide motivation for using existing checksum-based file systems (such as Sun's ZFS [32]). While their technique is more precise than random testing, it does not find the crash errors that EXPLODE does, nor is it as systematic. Extending EXPLODE to similarly return garbage on disk reads is trivial.

Software Model Checking. Model checkers have been previously used to find errors in both the design and the implementation of software systems [1, 3, 7, 13, 15, 16, 18, 30]. Two notable examples are Verisoft [13], which systematically explores the interleavings of a concurrent C program, and Java PathFinder [3] which used a specialized virtual machine to check concurrent Java programs by checkpointing states.

The model checking ideas EXPLODE uses — exhausting states, systematic exploration, and choice — are not novel. This paper's conceptual contribution is dramatically reducing the large work factor that plagues traditional model checking. It does so by turning the checking process inside out. It interlaces the control it needs for systematic state exploration *in situ*, throughout the checked system. As far as we know, EXPLODE is the first example of *in situ* model checking. The paper's engineering contribution is building a system that exploits this technique to effectively check large amounts of storage system code with relatively little effort.

Static bug finding. There has been much recent work on static bug finding (e.g., [1, 5, 8, 9, 11, 12]). Roughly speaking, because dynamic checking runs code, it is limited to just executed paths, but can more effectively check deeper properties implied by the code (e.g., `sync()` actually commits data to stable storage or crash recovery works). The errors we found would be difficult to get statically. However, we view static analysis as complementary: easy enough to apply that there is no reason not to use it and then use EXPLODE.

12 Conclusion and Future Work

EXPLODE comprehensively checks storage systems by adapting key ideas from model checking in a way that retains their power but discards their intrusiveness. Its interface lets implementors quickly write storage checkers, or simply compose them from existing components. These checkers run on live systems, which means they do not have to emulate either the environment or pieces of the system. As a result, we often have been able to check a new system in minutes. We used EXPLODE to find serious bugs in a broad range of real, widely-used storage systems, even when we did not have their source code. Every system we checked had bugs. Our gut belief has become that an unchecked system *must* have bugs — if we do not find any we immediately look to see what is wrong with our checker (a similar dynamic arose in our prior static checking work).

The work in this paper can be extended in numerous ways. First, we only checked systems we did not build. While this shows EXPLODE gets good results without a deep understanding of checked code, it also means we barely scratched the surface of what could be checked. In the future we hope to collaborate with system builders to see just how deep EXPLODE can push a valued system.

Second, we only used EXPLODE for bug-finding, but it is equally useful as an end-to-end validation tool (with no bug fixing intended). A storage subsystem implementor can use it to double-check that the environment the subsystem runs in meets its interface contracts and that the implementor did not misunderstand these contracts. Similarly, a user can use it to check that slipping a subsystem into a system breaks nothing. Or use it to pick a working mechanism from a set of alternatives (e.g., if `fsync` does not work use `sync` instead).

Finally, we can do many things to improve EXPLODE. Our biggest missed opportunity is that we do nothing clever with states. A big benefit of model checking is perspective: it makes state a first-class concept. Thus it becomes natural to think about checking as a state space search; to focus on hitting states that are most "different" from those already seen; to infer what actions cause "interesting" states to be hit; and to extract the essence of states so that two superficially different ones can be treated as equivalent. We have a long list of such things to add to EXPLODE in the future.

Acknowledgements

We thank Xiaowei Yang, Philip Guo, Daniel Dunbar, Silas Boyd-Wickize, Ben Pfaff, Peter Pawlowski, Mike Houston, Phil Levis for proof-reading. We thank Jane-Ellen Long and Jeff Mogul for help with time management. We especially thank Ken Ashcraft and Cristian Cadar for detailed comments, Jeremy Sugerman for his help in reasoning about the GSX error, and Paul Twohey and Ben Pfaff for help in the initial stages of this project (described in [31]). We thank Martin Abadi (our shepherd) and the anonymous reviewers for their struggles with our opaque submission. This research was supported by National Science Foundation (NSF) CAREER award CNS-0238570-001 and Department of Homeland Security grant FA8750-05-2-0142.

References

[1] T. Ball and S. Rajamani. Automatically validating temporal safety properties of interfaces. In *SPIN 2001 Workshop on Model Checking of Software*, May 2001.

[2] Berkeley DB. `http://www.sleepycat.com`.

[3] G. Brat, K. Havelund, S. Park, and W. Visser. Model checking programs. In *IEEE International Conference on Automated Software Engineering*, 2000.

[4] N. Brown. Private communication., Mar. 2005.

[5] W. Bush, J. Pincus, and D. Sielaff. A static analyzer for finding dynamic programming errors. *Software: Practice and Experience*, 30(7):775–802, 2000.

[6] E. Clarke, O. Grumberg, and D. Peled. *Model Checking*. MIT Press, 1999.

[7] J. Corbett, M. Dwyer, J. Hatcliff, S. Laubach, C. Pasareanu, Robby, and H. Zheng. Bandera: Extracting finite-state models from Java source code. In *ICSE 2000*, 2000.

[8] The Coverity software analysis toolset. `http://coverity.com`.

[9] M. Das, S. Lerner, and M. Seigle. Path-sensitive program verification in polynomial time. In *Proceedings of the ACM SIGPLAN 2002 Conference on Programming Language Design and Implementation*, Berlin, Germany, June 2002.

[10] G. Dunlap, S. T. King, S. Cinar, M. Basrat, and P. Chen. ReVirt: enabling intrusion analysis through virtual-machine logging and replay. In *Proceedings of the Fifth Symposium on Operating Systems Design and Implementation*, Dec. 2002.

[11] D. Engler, B. Chelf, A. Chou, and S. Hallem. Checking system rules using system-specific, programmer-written compiler extensions. In *Proceedings of Operating Systems Design and Implementation*, Sept. 2000.

[12] C. Flanagan, K. Leino, M. Lillibridge, G. Nelson, J. Saxe, and R. Stata. Extended static checking for Java. In *Proceedings of the ACM SIGPLAN 2002 Conference on Programming Language Design and Implementation*, pages 234–245. ACM Press, 2002.

[13] P. Godefroid. Model Checking for Programming Languages using VeriSoft. In *Proceedings of the 24th ACM Symposium on Principles of Programming Languages*, 1997.

[14] HFS and HFS+ utilities. `http://darwinsource.opendarwin.org/10.2.6/diskdev_cmds-208.11`.

[15] G. J. Holzmann. The model checker SPIN. *Software Engineering*, 23(5):279–295, 1997.

[16] G. J. Holzmann. From code to models. In *Proc. 2nd Int. Conf. on Applications of Concurrency to System Design*, pages 3–10, Newcastle upon Tyne, U.K., 2001.

[17] M. K. *Symbolic Model Checking*. Kluwer Academic Publishers, 1993.

[18] M. Musuvathi, D. Y. Park, A. Chou, D. R. Engler, and D. L. Dill. CMC: A pragmatic approach to model checking real code. In *Proceedings of the Fifth Symposium on Operating Systems Design and Implementation*, 2002.

[19] Linux NFS. `http://nfs.sourceforge.net/`.

[20] D. Patterson, G. Gibson, and R. Katz. A case for redundant arrays of inexpensive disks. *ACM SIGMOD Conference*, pages 109–116, June 1988.

[21] V. Prabhakaran, L. N. Bairavasundaram, N. Agrawal, H. S. Gunawi, A. C. Arpaci-Dusseau, and R. H. Arpaci-Dusseau. Iron file systems. In *Proceedings of the Twentieth ACM Symposium on Operating Systems Principles*, pages 206–220, New York, NY, USA, 2005. ACM Press.

[22] J. Saltzer, D. Reed, and D. Clark. End-to-end arguments in system design. *ACM Transactions on Computer Systems*, 2(4):277–288, Nov. 1984.

[23] Sandberg, Goldberg, Kleiman, Walsh, and Lyon. Design and implementation of the Sun network file system, 1985.

[24] A simple block driver. `http://lwn.net/Articles/58719/`.

[25] M. Sivathanu, V. Prabhakaran, F. I. Popovici, T. E. Denehy, A. C. Arpaci-Dusseau, and R. H. Arpaci-Dusseau. Semantically-smart disk systems. In *Second USENIX Conference on File and Storage Technologies*, 2003.

[26] Linux software RAID. `http://cgi.cse.unsw.edu.au/~neilb/SoftRaid`.

[27] Subversion. `http://subversion.tigris.org`.

[28] J. Sugerman. Private communication., Dec. 2005.

[29] VMware GSX server. `http://www.vmware.com/products/server/`.

[30] J. Yang, P. Twohey, D. Engler, and M. Musuvathi. Using model checking to find serious file system errors. In *Proceedings of the Sixth Symposium on Operating Systems Design and Implementation*, Dec. 2004.

[31] J. Yang, P. Twohey, B. Pfaff, C. Sar, and D. Engler. eXplode: A lightweight, general approach for finding serious errors in storage systems. In *Workshop on the Evaluation of Software Defect Detection Tools*, June 2005.

[32] Zfs: the last word in file systems. `http://www.sun.com/2004-0914/feature/`.